Implementation of a Distributed Commit/Termination Protocol by Communicating Moore Machines

Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602

86-60-10

Implementation of a Distributed Commit/Termination Protocol by Communicating Moore Machines

Toshimi Minoura

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602

ABSTRACT

In this short paper, we present an implementation method for a *distributed commit/termination protocol* for a distributed database system. The protocol, which handles both commit and termination processing of distributed transactions, is represented by *communicating Moore machines.* Several advantages of our approach are discussed.

Key Words and Phrases: distributed database system, atomicity, commit/termination protocol, communicating Moore machine.

1. Introduction

In a distributed database system, a transaction may create updates at multiple sites. Then all of those updates must be applied to stored data, or all of them must be discarded. This requirement is known as the *atomicity* requirement [LAMP-81].

The *two-phase commit protocol* (2PC) is the best known protocol that preserves the atomicity of distributed transactions [LAMP-81, GRAY-78]. However, 2PC is a *centralized* commit/termination protocol, and it possesses the following problem. The *coordinator* of 2PC unilaterally makes a commit/termination decision for a distributed transaction. Hence, when the coordinator becomes inaccessible from some subtransactions because of site or communication failures in the system, the commit/termination processing of those subtransactions may have to be blocked until the communication with the coordinator is restored.

In order to reduce the likelihood of such blocking, a series of *decentralized* commit/termination protocols were devised by Skeen [SKEE-81a, SKEE-81b, SKEE-82, SKEE-83]. In this short paper, we present a simple implementation method for a decentralized commit/termination protocol. Although the protocol is in its essence an integration of Skeen's protocols, it incorporates several improvements.

1. The entire commit/termination protocol is represented by *communicating Moore machines.* Each Moore machine needs to know only the current states of other Moore machines, and messages are not part of the system state. Further, late state messages do not do any harm in our protocol. Hence, each Moore machine needs to broadcast only its current state whenever the system is reconfigured. Thus, the implementation of the protocol is easy.

2. Since the entire protocol is represented in a precise formalism of communicating Moore machines, correctness of the protocol can be checked mechanically.

3. We represent subtransactions and coordinators by different Moore machines. Therefore, even if the number of subtransactions created by each ditributed transaction varies, the number of coordinators used by each distributed transaction can be identical. Also, the separation of coordinators and subtransactions makes the protocol easy to comprehend.

4. We separate the basic protocol from policies. The basic protocol alone guarantees the atomicity of transactions. As long as the basic protocol is observed, different policies can be implemented in order to meet different requirements for system availability and ease of implementation.

5. Since commit processing and termination processing are handled by one protocol, switching of protocols for those two purposes is not required. Actually, our protocol does not require site failures or network partitioning to be detected in order for it to work correctly. The only requirement is that possibly lost state

messages should be retransmitted. This requirement may be satisfied by a simple timeout mechanism.

Skeen proposed to model a commit protocol by communicating sequential machines [SKEE-83], and protocols modeled according to his formalism have been extensively analyzed [CHEU-85, CHEU-86, CHIN-83]. Hammer and Shipman proposed a scheme in which coordinators are separated from subtransactions as *backup processes* [HAMM-80]. However, their backup processes are different from our coordinators in several respects.

2. Communicating Moore Machines

In describing our protocol, we will use a *system of communicating Moore machines* (CMMs). A key feature of CMMs is that messages exchanged among them carry only information that can be reproduced from the state information of the participating CMMs. This fact makes analysis and implementation of our protocol easy.

Each CMM consists of a set S of *states*, a set T of *transitions* ($T \subseteq S \times S$), and an *enabling function* e. One state in S is the *initial state* s_0 of the CMM. Enabling function e associates with each transition t in T a set $e(t)$ of multisets of states of CMMs. A multiset in $e(t)$ is a submutiset of the multiset S' of the states of all the CMMs in the system. Starting with the initial state s_0 and following *enabled* transitions, a CMM is allowed to make state transitions. The *current* state of a CMM is its latest state reached. A transition t is enabled if all the states in some mutiset in $e(t)$ are current. A multiset in $e(t)$ is called an *enabling condition* of t.

In the above definition of a system of CMMs, only the current states of the CMMs are used in deciding enabled transitions. However, if the current state of a CMM M is propagated to other CMMs by messages, those messages may arrive at other CMMs after M has changed its state and may cause an unintentional state transition. We say that a system of CMMs is *impervious to obsolete state messages* if every state transition enabled by old state messages is enabled by the current states of the CMMs. We will show in the Section 4 that our protocol is impervious to obsolete state messages.

3. Protocol

We now show our protocol as a collection of CMMs. A transaction creates a *subtransaction* at each site where it accesses stored data. The *coordinators* of each transaction T are responsible for making the effect of T *atomic*. We assume that state information of each subtransaction or coordinator is reliably stored at a site in the system, and that it will be left intact even if a site or communication failure occurs.

It is desirable that even if some coordinators of a transaction become isolated, others can make a consistent decision about the fate of the transaction. For this purpose, we use a simple *voting* scheme. Assume that a transaction creates N_t subtransactions that are managed by N_c coordinators. A transaction can be *committed* if at least V_c coordinators agree to do so, and it can be *aborted* if at least V_a coordinators agree to do so. Now, if $V_c + V_a > N_c$, no conflicting decisions will be made. Since it is desirable that a group of coordinators as small as possible can make a decision, we let $V_c + V_a = N_c + 1$.

Fig. 1 shows a CMM for each subtransaction, and Fig. 2 a CMM for each coordinator. The enabling conditions of each transition t are given next to the arrow representing t. For example, in Fig. 2, $e((w,u)) = \{\{\mathrm{p}(N_t)\}, \{\mathrm{u}\}\}$, where $p(N_t)$ stands for N_t instances of p, indicating that transition (w,u) is enabled if N_t subtransactions are in state p or if at least one coordinator is in state u.

Normal processing of a transaction proceeds as follows.

1. When a subtransaction is started, it stays in state s (*start*) until N_c coordinators are created.

2. The initial state of each coordinator is w (*wait*).

3. Once all of the N_c coordinators are created, each subtransaction changes its state from s to r (*running*). While a subtransaction is in state r, it can read stored data and create *tentative* updates.

4. Once a subtransaction completes its processing, it enters state p (*prepared*).

5. When all of the N_t subtransactions enter state p, a coordinator can enter state u (*vote to commit*).

6. When at least V_c coordinators vote to commit, a coordinator can enter state c (*commit*).

7. Once any coordinator enters state c, other coordinators and subtransactions can enter state c.

8. When a subtransaction enters state c, it can make its updates permanent.

When normal processing of a transaction becomes impossible, the transaction may be aborted.

1. A subtransaction in state s or r can unilaterally enter state a (*abort*).

2. A coordinator in state w can enter state v (*vote to abort*).

3. When at least V_a coordinators vote to abort, a coordinator can enter state a.

4. Once any sutransaction or coordinator enters state a, other coordinators and subtransactions can enter state a.

Our protocol represented as a collection of CMMs is nondeterministic, and it does not tell how soon an enabled transition should be activated. However, it is desirable to exercise certain discretion in selecting a transition to be activated, especially one that leads to a transaction abortion. A *policy* is a rule stipulated for this purpose.

We recommend the following set of policies.

1. Transitions other than those from *s* to *a*, from *r* to *a*, from *w* to *u*, and from *w* to *v* should be activated as soon as they are enabled.

2. A subtransaction must move from *s* or *r* to *a* if it knows that the transaction to which it belongs cannot be completed or if it is suspended in state *s* for an unreasonably long time because some coordinators become inaccessible.

3. If the transition from *w* to *u* is enabled while a transaction is being processed normally, a coordinator should make that transition immediately.

4. If a coordinator is found in state *w* after the system is reconfigured, the coordinator must try to establish the quorum of either state *u* or *v*, consulting with other reachable coordinators. If neither of the quorums can be established, it should stay in state *w*. Note that a state transition from *w* to *v* can be freely made, but that one from *w* to *u* can be made only if it is enabled.

As we stated, the state information of CMMs must be reliably stored at sites. When a CMM moves to a new state, this state change is first recorded on main memory, and then the information must be moved to reliable storage. When a processor failure occurs, the state information stored on main memory is likely to be lost, and then it must be restored from reliable storage. When such restoration of the state information occurs, the states of CMMs may be rolled back. If a state change of a CMM is broadcast to other sites before the state change is recorded on reliable storage, messages carrying that state change information may have to be recalled. In order to avoid this difficulty, messages informing a state change should be generated only after that state change is recorded on reliable storage.

4. Correctness

We now want to show that the CMMs given in Figs. 1 and 2 guarantee the atomicity of a transaction. The effect of a transaction is atomic if all the coordinators and subtransactions terminate unanimously either in state c or in state a. It is possible to show this fact by informal reasoning, which we leave to the reader as an exercise. If the number of coordinators involved is few, the proof can be performed by drawing a complete state transition diagram.

In drawing a state transition diagram, however, a care must be taken since we allow old messages to cause state transitions. In order that all past states of each coordinator can be considered in deciding enabled transitions, we show the state history of each coordinator where we usually show only its current state.

Fig. 3 is the state transition diagram for a system consisting of three coordinators given in Fig. 2, where $V_c = V_a = 2$. It is assumed that all subtransactions have entered state p, and the diagram shows only the states of the coordinators.

Our protocol is impervious to old messages; i.e., all the transitions that may be enabled by old state messages ((u,c) and (v,c) in Fig. 2) are enabled also by the current states of the CMMs. Consider the system state $\{wu, wuc,$ wuc$\}$, which indicates the system state after one coordinator moved from state w to state u and the other two coordinators moved from state w to state u and then to state c. At this point the enabling condition $u(V_c)$ for the transition from u to c of the coordinator whose current state is u is not satisfied by the current states of the coordinators since only one coordinator is in state u, but this enabling condition is satisfied by old states. Nonetheless, this fact does not make any difference since the other enabling condition (c) assigned to the same transition is satisfied anyway.

According to the state transition diagram, all the coordinators terminate unanimously either in state c or in state a. Hence, our protocol guarantees the atomicity of a transaction.

A policy for selecting transitions to be activated can be implemented freely as long as the basic protocol shown in Figs. 1 and 2 are observed. However, in order to guarantee a proper termination of the protocol, the following rule must be observed in such a policy. In any system state, if there exisits enabled transitions, at least one of them must be activated eventually.

It is easy to show that all the coordinators and subtransactions will eventually terminate when the above rule is observed. All the CMMs involved are acyclic, and they can proceed only towards state *c* or *a*. If at least one transition takes place eventually in any non-terminal system state, every CMM eventaully reaches state *c* or *a*.

The fact that old state messages do not cause any harm to our protocol has another important implication. In the preceding section we stated that messages informing a state change must be generated after that state change is recorded on reliable storage. This requirement does not cause any adverse effect since state change messages can be delayed arbitrarily.

5. Discussion

In this section we consider when a group of coordinators fail to reach an agreement about the fate of the transaction managed by them. The main results are the following.

1. If the system is reconfigured only once while a transaction is being processed, the coordinators in a large enough group can terminate the transaction.
2. If the system is reconfigured more than once while a transaction is being processed, there is no fixed minimum for the number of coordinators that can terminate the transaction.

In the following discussion, we assume that the policies given in Section 3 are observed.

If the system is reconfigured only once while a transaction is being processed normally. The possible states for the coordinators are *w*, *u* and *c*. Assume that a group of coordinators is formed at this point. If at least one coordinator in the group is in state *c*, all the coordinators in the group can be moved to state *c*. If at least one coordinator in the group is in state *u* and if the size of the group is at least V_c, all the coordinators in the group can be moved to state *c* after some coordinators are moved from state *w* to state *u*. If all the coordinators in the group are in state *w* and if the size of the group is at least V_a, all the coordinators in the group can be moved to state *v* and then to state *a*. Thus, if a system reconfiguration occurs only once, any group of coordinators with its size at least V_c or V_a, whichever is greater, can reach an agreement. A similar result has been obtained for a Skeen's protocol by a complex analysis [CHEU-86].

A group of coordinators formed while the system is being reconfigured may not be able to reach an agreement even if its size is at least $max(V_c, V_a)$. Consider the following scenario. The system is being reconfigured after some coordinators have entered state *u*, and a group of coordinators whose size is at least V_a is formed. All the coordinators in this group happen to be still in state *w* and agree to abort the transaction. However, after some of the coordinators in the group enter state *v*, another system reconfiguration becomes necessary. Then, if a new group whose size is at least $max(V_c, V_a)$ is formed, some coordinators in it may be in state *u*, and others may be in state *v*. If such a situation occurs, the decision must be postponed until a larger group is formed where an enough number of state *u*'s or *v*'s can be found. Even in the worst case the coordinators can reach an agreement when all of them can communicate with each other again.

6. Conclusion

We presented an implementation method of a distributed commit/termination protocol for a distributed database system. The protocol was designed as communicating Moore machines, and several advantages of our approach were discussed.

References

[CHEU-85]
Cheung, D., and Kameda, *T.* Site optimal termination protocols for a distributed database under network partitioning Proc. 4th ACM Symp. on Principles of Distributed Computing, Aug. 1985, pp. 111-121.

[CHEU-86]
Cheung, D., and Kameda, *T.* Optimal decentralized termination protocols for partition failures. TR. 86-1, Laboratory for Computer and Communications Research, Simon Fraser Univ., Jan. 1986.

[CHIN-83]
Chin, F., and Ramarao, K.V.S. Optimal termination protocols for network partitioning. Proc. 2nd ACM Symp. on Principles of Database Systems, March 1983, pp. 25-35.

[GRAY-78]
Gray, J. Notes on data base operating systems. In *Lecture Notes in Computer Science 60,* Springer-Verlag, 1978, pp. 393-481.

[HAMM-80]
Hammer, M. and Shipman, D. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Tr. on Database Systems 5,* 4 (Dec. 1980), 431-466.

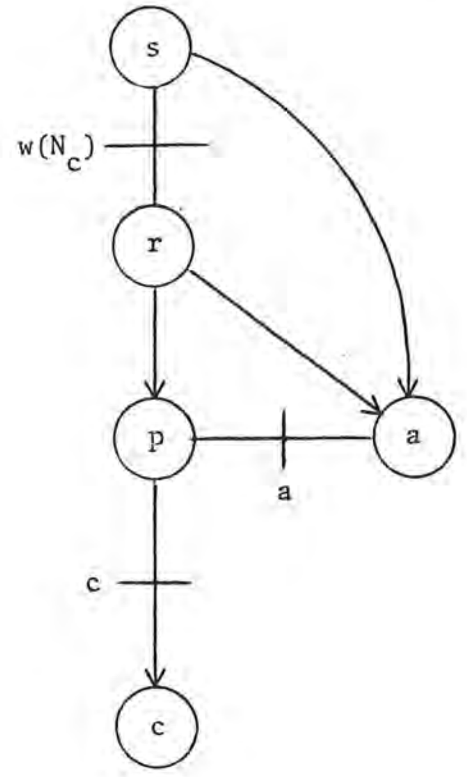
[LAMP-81]
Lampson, B.W. Atomic transactions. In *Distributed Systems -- Architecture and implementation,* Lecture Notes in Computer Science 105, Springer-Verlag, 1981, pp. 246-265.

[SKEE-81a]
Skeen, D. Nonblocking commit protocols. Proc. ACM-SIGMOD Intl. Conf. on Management of Data, 1981, pp. 133-142.

[SKEE-81b]
Skeen, D. A decentralized termination protocol. Proc. IEEE Symp. on Reliability in Distributed Software and Database Systems, 1981, pp. 27-32.

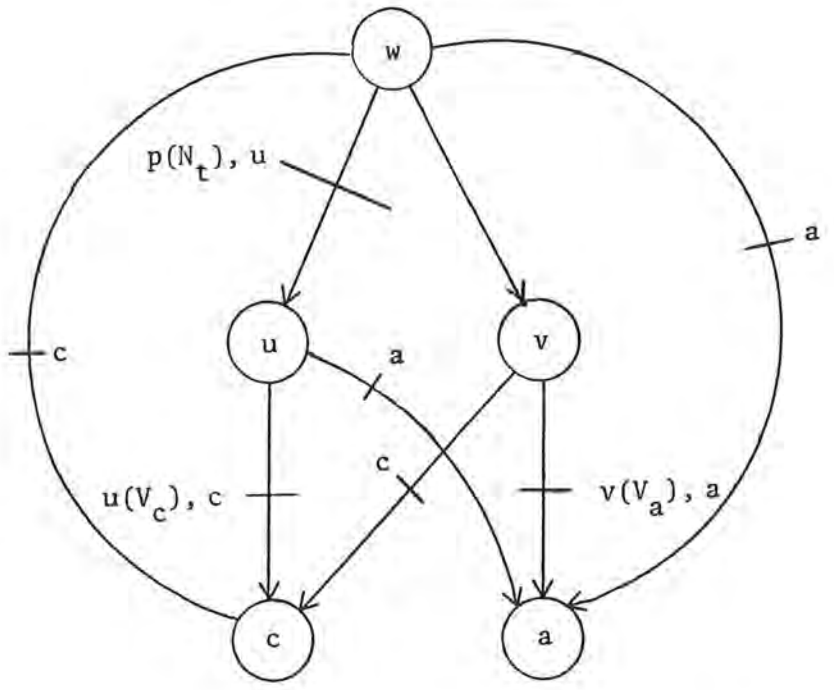
[SKEE-82]
Skeen, D. A quorum-based commit protocol. Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks, 1982, pp. 69-80.

[SKEE-83]
Skeen, D., and Stonebraker, M. A formal model of crash recovery in a distributed system. *IEEE Tr. on Software Engineering SE-9,* 3 (May 1983), 219-228.



s:	start
r:	running
p:	prepared
a:	abort
c:	commit

Fig. 1. A CMM for a participant.



w:	wait
u:	propose to commit
v:	propose to abort
c:	commit
a:	abort

Fig. 2. A CMM for a coordinator.

$$\begin{tikzcd}
 & & & & \{w,w,w\} \arrow[dll] \arrow[drr] & & & & \\
 & & \{w,w,wu\} \arrow[dl] \arrow[drr] & & & & \{w,w,wv\} \arrow[dll] \arrow[dr] & & \\
 & \{w,wu,wu\} \arrow[dl] \arrow[dd] \arrow[drr] & & & \{w,wu,wv\} \arrow[dl] \arrow[dr] & & & \{w,wv,wv\} \arrow[dll] \arrow[dd] \arrow[dr] & \\
\{wu,wu,wu\} \arrow[dd] & & & \{wu,wu,wv\} \arrow[dd] & & \{wu,wv,wv\} \arrow[dd] & & & \{wv,wv,wv\} \arrow[dd] \\
 & \{w,wu,wuc\} \arrow[dl] \arrow[ddd] \arrow[ddr] \arrow[drr] & & & & & & \{w,wv,wva\} \arrow[dll] \arrow[ddl] \arrow[ddd] \arrow[dr] & \\
\{wu,wu,wuc\} \arrow[dddd] & & & \{wu,wuc,wv\} \arrow[ddddl] \arrow[ddddd] & & \{wu,wv,wva\} \arrow[ddddd] \arrow[ddddr] & & & \{wv,wv,wva\} \arrow[dddd] \\
 & & \{wu,wc,wuc\} \arrow[dd] & & & & \{wv,wa,wva\} \arrow[dd] & & \\
 & \{w,wuc,wuc\} \arrow[ddl] \arrow[dr] \arrow[ddr] & & & & & & \{w,wva,wva\} \arrow[dl] \arrow[ddl] \arrow[ddr] & \\
 & & \{wc,wuc,wuc\} & & & & \{wa,wva,wva\} & & \\
\{wu,wuc,wuc\} \arrow[dd] & & \{wuc,wuc,wv\} \arrow[ddr] & & & & \{wva,wva,wu\} \arrow[ddl] & & \{wv,wva,wva\} \arrow[dd] \\
 & & & \{wu,wuc,wvc\} \arrow[d] & & \{wv,wva,wua\} \arrow[d] & & & \\
\{wuc,wuc,wuc\} & & & \{wuc,wuc,wvc\} & & \{wva,wva,wua\} & & & \{wva,wva,wva\}
\end{tikzcd}$$

Fig. 3. State transition diagram.